Theoretical Computer Science 429 (2012) 202–212

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Computation as an unbounded process

Jan van Leeuwen [a,*], Jiří Wiedermann [b]

[a] Department of Information and Computing Sciences, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands

[b] Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic

ARTICLE INFO

Keywords:
Arithmetical Hierarchy
Hypercomputation
Mind change complexity
Nondeterminism
Relativistic computation
Unbounded computation

ABSTRACT

We develop a model of computation as an unbounded process, measuring complexity by the number of observed behavioural changes during the computation. In a natural way, the model brings effective unbounded computation up to the second level of the Arithmetical Hierarchy, unifying several earlier concepts like trial-and-error predicates and relativistic computing. The roots of the model can be traced back to the circular a-machines already distinguished by Turing in 1936. The model allows one to introduce nondeterministic unbounded computations and to formulate an analogue of the *P*-versus-*NP* question. We show that under reasonable assumptions, the resource-bounded versions of deterministic and nondeterministic unbounded computation have equal computational power but that in general, the corresponding complexity classes are different ($P^{mind} \subsetneq NP^{mind}$).

1. Introduction

Computation is arguably the most basic form of information processing. In the philosophy of computation, it is often understood as transforming information by some repeated systematic process, without constraining this a priori much further. For example, Goldreich [9] describes computation as 'a process that modifies an environment by the repeated application of a predetermined simple rule', and suggests that it can apply equally to a wide variety of processes in 'natural reality' and to the human-defined or -created processes in 'artificial reality', where he associates the latter with the world of computers and automation.

It follows that there are many more conceivable starting points for modelling computation than the time-tested approaches initiated more than seventy years ago by Turing [24] and his contemporaries, departing from the intuitive notion of a computer (human or machine) that computes function or numbers in finite time. In particular, computational mechanisms today aim at a rather different and broader range of tasks and feature properties like interactivity (with external agents), non-uniformity of program (learning and extension) and unbounded operation over time instead, as crucial characteristics of their functioning [26]. In this paper, we make a special study of *computation as an unbounded process*.

1.1. Model

In typical computer applications nowadays, the core mechanism is a *multi-process system* that is always up and running. Control goes from process to process and, whenever a process has its turn, the process computes until it executes an instruction that explicitly transfers control to another process. It is this mechanism that we will study and appraise computationally. As we will be interested in tracing the computation globally, we will delineate one process as the 'base

* Corresponding author.
E-mail addresses: J.vanLeeuwen@cs.uu.nl (J. van Leeuwen), Jiri.Wiedermann@cs.cas.cz (J. Wiedermann).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.12.040

Table 1
Approaches to unbounded computation.

Model of computation	Level	Reference	year
Non-terminating circular a-machines	Σ_2, Π_2	Turing [24]	1936
Number-theoretic predicates	Arithmetic sets	Kleene [14]	1943
Oracle Turing machines	Arithmetic sets	Turing citeTuring1939, Post [16]	1939
Trial-and-error predicates	Δ_2	Putnam [18]	1965
Limiting recursion	Δ_2	Gold [7]	1965
Iterated limiting recursion	Δ_k	Schubert [21]	1974
Alternating Turing machines	Arithmetic sets	Chandra et al. [2]	1976
ω-Turing machines	NA	Cohen & Gold [3]	1978
Tae-computability	Σ_2	Hintikka & Mutanen [12]	1998
Infinite time Turing machines	Hyperarithmetic sets	Hamkins & Lewis [11]	2000
Accelerating Turing machines	NA	Copeland [4]	2002
Relativistic computing	Δ_2, Σ_2	Etesi & Nemeti [6,29]	2002
SAD computers	Arithmetic sets	Hogarth [13]	2004
Zeno machines	NA	Potgieter [17]	2006
Display Turing machines	Δ_3	Rovan & Steskal [20]	2007
Red–green Turing machines	Σ_2, Π_2, Arithmetic sets	*this paper*	2010

process' (presumably the process in which the machine will attempt to 'stabilize' after any internal run) and all other processes as 'transient processes' (presumably meant as processes for passing through some subtasks).

Abstracting further, the core mechanism can be viewed as consisting of two processes: the 'green process' (the base process) and the 'red process' (the collective transient processes), and control switching between the red and the green process as dictated by the computation inside the processes. Using the Turing machine as basic underlying architecture, this leads to the notion of *red–green Turing machine*, which is an ordinary offline Turing machine with the internal states partitioned into red states and green states. The red–green machine has a 2-way read-only input tape and an unbounded, i.e. potentially infinite work tape as usual. Computations on a red–green machine begin with a finite word w on the input and the input head positioned on the left-most square of the input tape, and the machine in a fixed initial state which is **red**. After starting, the red–green machine computes *ad infinitum*, modelling the action of the red and green processes. Without loss of generality we assume that the machine never blocks.

To complete the model, we need a way to *observe* the behaviour and progress of the computation. Instead of counting time or space, we will keep track of the system by the number of *process switches*. We do so by logging these switches on a special one-way, write-only output channel: the machine writes 'red' when it switches to a red state, and green when it switches to a green state. (Recall that the machine always begins in a red state.) The machine does not log anything as long as it stays within states of the same colour. Consequently, the last colour written on the output always tells the colour of the state we are currently in.

Switching from red to green or vice versa can also be interpreted as a *mind change* of the machine. The notion of mind change is derived from the theory of machine learning, and we will see several connections with this later on [8]. The terminology was also used by Putnam [18] in developing his notion of *trial and error predicates*, which are a source of inspiration for this paper. Mind changes are a form of (weak) complexity measure, as we expect a machine to more quickly reach some conclusion the fewer mind changes it needs for it. At the same time, we will see that red–green TMs are a concrete model for the notion of relativistic computing [6,29,28], which gives the latter a more concrete place in computing. Table 1 shows the range of alternative approaches to unbounded computation which have been proposed over the years and which, in many cases, are easily simulated by red–green TMs.

Most interesting, however, is the fact that the notion of computation as an unbounded process as we distinguished it can be traced back to the discussion of the so-called *automatic machines* (or: *a-machines*) in Turing's fundamental 1936 paper [24]. Whereas the vocabulary and notational style were different, we will argue that the non-terminating version of, what Turing called, circular a-machines, coincides quite precisely with red–green computation [27]. Our study thus seems to fill a gap that existed since then.

1.2. Concepts

We first consider deterministic red–green machines.

Definition 1. A Turing machine $\mathcal{M}$ is called a *red–green Turing machine* if its set of internal states $\mathcal{Q}$ is partitioned into two subsets, $\mathcal{Q}_r$ and $\mathcal{Q}_g$, and $\mathcal{M}$ operates without halting. $\mathcal{Q}_r$ is called the set of *red states*, $\mathcal{Q}_g$ the set of *green states*. The initial state of $\mathcal{M}$ is assumed to be *red*.

To quantify the study of red–green machines, we will make use of the framework of recognizable sets. The following definition relates to the notion of *limiting recursion* due to Gold [7], extending it to the framework of machines.

Definition 2. Let $\mathcal{M}$ be a red–green TM.

(i) A word $w \in \Sigma^*$ is said to be *recognized* by $\mathcal{M}$ in (precisely) k mind changes if the unbounded computation of $\mathcal{M}$ on input w eventually stabilizes in green (i.e. stays in green states forever) after having made exactly k mind changes.
(ii) A word $w \in \Sigma^*$ is said to be *recognized* by $\mathcal{M}$ if w is recognized by $\mathcal{M}$ in some finite number of mind changes.
(iii) The set of strings (language) $L \subseteq \Sigma^*$ *recognized* by $\mathcal{M}$ is the set $L = L_{\mathcal{M}} = \{w|$ w is recognized by $\mathcal{M}\}$.

Definition 3. A set of strings $L \subseteq \Sigma^*$ is said to be recognizable within k (or, a bounded number of) mind changes if and only there exists a red–green TM $\mathcal{M}$ that recognizes every word $w \in L$ within at most k (or, a fixed bounded number of) mind changes.

In the definitions we might also wish to speak of *sets recognizable in the limit*, as the red–green TM operates forever and we may not know for certain by just observing it for a finite time when it has stopped 'changing its mind'. Also, the strings recognized by $\mathcal{M}$ are not necessarily all recognized within the same number of mind changes.

A red–green TM can *reject* a string w in two ways: either it eventually stabilizes in red, or does not stabilize on any colour at all (thus making an unbounded number of mind changes as the computation proceeds). We make a difference between red–green TMs that 'merely' operate as recognizer and those that operate as acceptor. In the latter case, we assume that the machine does *not* admit any computations in which it makes infinitely many mind changes.

Definition 4. Let $\mathcal{M}$ be a red–green TM. The set of strings (language) $L \subset \Sigma^*$ is said to *accepted* by $\mathcal{M}$ if the following two conditions are satisfied:

(a) $L = L_{\mathcal{M}} = \{w|$ w is recognized by $\mathcal{M}\}$.
(b) For every word $w \notin L$, the computation of $\mathcal{M}$ on input w eventually stabilizes in red. (In this case we say that w is rejected.)

It is easy to see that if a set $L \subseteq \Sigma^*$ is accepted by a red–green TM, then so is its complement. Red–green TMs that accept (or reject) their inputs will be called *red–green TM acceptors*. An equivalent of Lemma 1 does not clearly hold for acceptors (as one may have to preserve blocks of red states). The following fact follows easily. (Recall that a 2-r.e. set is any set that is the difference of 2 recursively enumerable sets.)

Lemma 1. *Let $\mathcal{M}$ be a (deterministic) red–green TM, $L \subseteq \Sigma^*$ a set of strings.*

(i) *If L is recognizable by $\mathcal{M}$ within a computably bounded number of mind changes, then L is acceptable by a red–green TM.*
(ii) *Let L_k (necessarily k odd) be the set of strings recognized by $\mathcal{M}$ in exactly k mind changes. Then L_k is a* 2*-r.e. set.*

Proof. Part (i) follows by first computing the bound and then keeping count of the number of mind changes with a separate subroutine of the machine, in a separate counter. The bound can be used to limit the number of mind changes needed to make non-accepting computations rejecting, e.e. convergent to red. Part (ii) follows by writing $L_k = A_k - A_{k+1}$, where A_k is the set of strings w on which $\mathcal{M}$ makes at least k mind changes. A_k is recursively enumerable for every $k \geq 1$ (modify $\mathcal{M}$ so it stops in a final state exactly when it makes the k-th mind change). □

No further details are needed to explain red–green TMs. They are perhaps the simplest model for unbounded computation, rooted in multi-process computation. We will show in Section 2 that red–green TMs recognize precisely the sets in Σ_2 sets and accept precisely the sets in Δ_2. Red–green TMs also allow for a direct interpretation of the Ershov hierarchy [5] and a machine proof of Post's theorem for Δ_2 ([15], Thm IV.1.16). Thus, the simple model of multi-process system computation brings us already in a natural way up to the second level of the Arithmetical Hierarchy, well beyond the domain of classical effective computation and yielding a recursion-theoretic view of many approaches to *hypercomputability*. For preliminaries from recursion theory we refer to [15,19].

1.3. Results

A major aim of this study is to analyse the effect of nondeterminism on mind change complexity. Importantly, the model of red–green TMs allows us to introduce *nondeterminism* in a natural way into the realm of unbounded computations. All we need is to allow the underlying TM to be nondeterministic. We may assume w.l.o.g. that a nondeterministic red–green TM always has one or two options available per instruction, never more or less.

Definition 5. Let $\mathcal{M}$ be a nondeterministic red–green TM. A word $w \in \Sigma^*$ is said to be *recognized* by $\mathcal{M}$ in (precisely) k mind changes if at least one of $\mathcal{M}$'s unbounded computations on input w eventually stabilizes in green (i.e. stays in green states forever) after having made exactly k mind changes.

The other definitions related to recognition, and the recognized set of strings by $\mathcal{M}$ remain unchanged. Also the notion of recognizability in the limit carries over to the nondeterministic case in the usual form.

We will be interested in the difference in quality between deterministic and nondeterministic red–green TMs, in terms of the number of mind changes they may need towards a certain goal. It leads to the following definitions, which maybe be indexed by any special subclass of red–green TMs one likes to consider.

Definition 6. Consider the operation of red–green TMs, let $f : \mathbb{N} \to \mathbb{N}$ be a function.

(i) $MIND[f]$ is the class of languages recognized (accepted) by deterministic red–green TMs within a number of mind changes bounded by $f(n)$.
(ii) $ND\_MIND[f]$ is the class of languages recognized (accepted) by nondeterministic red–green TMs within a number of mind changes bounded by $f(n)$.

It will clear what is meant by $MIND[\mathcal{F}]$ and $ND\_MIND[\mathcal{F}]$ for suitable classes of functions $\mathcal{F}$. For example, we may be interested in polynomially bounded simulations, if they exist. (We will also be interested in the relation between deterministic and nondeterministic convergence in the limit.)

Definition 7. Consider the operation of red–green TMs.

(i) $P^{mind} (= MIND[poly])$ is the class of languages recognized (accepted) by deterministic red–green TMs within a polynomially bounded number of mind changes.
(ii) $NP^{mind} (= ND\_MIND[poly])$ is the class of languages recognized (accepted) by nondeterministic red–green TMs within a polynomially bounded number of mind changes.

While nondeterministic red–green TMs can always be simulated effectively by deterministic red–green TMs, we will prove:

Theorem A. *There is no computable function f such that, if L is recognized by a nondeterministic red–green TM within k mind changes, then L can be recognized by a deterministic red–green TM within $f(k)$ mind changes.*

We will show that there are various reasonable types of nondeterministic red–green TMs that *can* be simulated efficiently, i.e. which have behaviours that can be simulated deterministically without needing too many extra mind changes for it. Nevertheless, a typical side result of our analysis will be:

Theorem B. $P^{mind} \neq NP^{mind}$.

The model of red–green TMs makes computation-in-the-limit accessible for an analysis of the concrete computational behaviours of this phenomenon, under a variety of typical constraints as they are known in other models of computation.

2. Basic facts for red–green TMs

In this section we summarize a number of basic facts for red–green TMs. The observations capture the recognition capabilities of the machine and are implicit in several related results, but they find a slightly generalized form here.

2.1. Definitional remarks

The definition of red–green TMs was motivated by the operation of multi-process systems. On the other hand, red–green TMs can be seen as a type of ω-TM on finite inputs with a recognition criterion based on some property of the set(s) of states visited (in)finitely often, in the tradition of ω-automata (cf. [23]). In particular, red–green TMs precisely correspond to ω-TM with a *Rabin-type recognition criterion*: an infinite run of the TM on input w is recognizing if and only if (a) no red state is visited infinitely often and (b) some green states (one or more) are visited infinitely often.

Most interesting, however, is the fact that the notion of computation as an unbounded process as we distinguished it can be traced back to the discussion of the so-called *automatic machines* (or: *a-machines*) in Turing's 1936 paper [24]. Properly designed a-machines could well be *circle-free*, e.g. if their task would be to output infinite sequences of binary digits representing computable reals. Turing noted that there may be machines, which he called *circular*, which at some point stop producing outputs, i.e. which altogether produce only a finite number of output bits. Whereas a machine may be circular because it stops after finitely many steps, there is the option of being circular while not terminating. Omitting the formal details, we quote from [27]:

Theorem 1. *Let A be a non-terminating circular a-machine, let $\mathcal{M}$ be a red–green Turing machine, let $f : \{0,1\}^* \to \{0,1\}^*$ be a partial function. Then f is computed by $\mathcal{A}$ if and only if f is computed by $\mathcal{M}$ performing $f(x)$ mind changes.*

2.2. Computational power of red–green TMs

Red–green TMs certainly have a far greater reach in computational power than classical TMs. At the base however, there is the power of ordinary TMs.

Lemma 2. *A set of strings L is recognized by a red–green TM within one mind change if and only if $L \in \Sigma_1$, i.e. if L is recursively enumerable.*

Proof. Let L be the set of strings recognized by a red–green TM $\mathcal{M}$ within one mind change. L is seen to be r.e.: design a TM that enumerate all possible inputs, simulates and dovetails the computations of $\mathcal{M}$ on these inputs, and outputs string w whenever $\mathcal{M}$ makes its first mind change (if any) during the computation on w.

Conversely, if $L \in \Sigma_1$ and $\mathcal{N}$ is the TM that enumerates L, then design a red–green TM that on input w simulates the computation of $\mathcal{N}$ in red but switches to green when w appears in the enumeration. The machine precisely recognizes L. □

If more mind changes are allowed, the full power of red–green machines is unleashed. Note e.g. that the complement of every r.e. set L, which is not necessarily r.e. again, is always red–green recognizable. For, let TM $\mathcal{N}$ recognize L. Design a red–green TM $\mathcal{M}$ that operates on inputs w as follows: starting in red, the machine immediately switches to green and starts simulating $\mathcal{N}$ on w. If $\mathcal{N}$ halts (thus recognizing w), the machine switches to red and stays in red from then onward. It follows that $\mathcal{M}$ precisely recognizes, in fact accepts, the set $\bar{L}$.

With various pieces of evidence from existing theory, the following result offers itself as a characterization of the computational power of red–green TMs in general. Part (ii) is related to Putnam's characterization of trial and error predicates ([18], Theorem 1).

Theorem 2. *Consider red–green TMs.*

(i) *Red–green Turing machines recognize exactly the Σ_2 sets of the Arithmetical Hierarchy.*
(ii) *Red–green Turing machines accept exactly the Δ_2 sets of the Arithmetical Hierarchy.*

Proof. Let L be the language recognized by an red–green TM $\mathcal{M}$. Design a recursive predicate $F(x, y, w)$ with the following semantics:

> with x denoting an integer, if y represents the computation of $\mathcal{M}$ on input w for a number of steps that is greater than or equal to x, then y proceeds only in green states from step x onward.

Such a predicate clearly exists. It follows that $L \in \Sigma_2$ because $w \in L \Leftrightarrow \exists_x \forall_y F(x, y, w)$. If $\mathcal{M}$ accepts language L, then the complement machine recognizes $\bar{L}$. It follows in this case that both $L \in \Sigma_2$ and $\bar{L} \in \Sigma_2$, thus $L \in \Pi_2$, hence $L \in \Delta_2$.

Conversely, if $L \in \Sigma_2$, then there exists a recursive predicate $F(x, y, w)$ such that $w \in A \Leftrightarrow \exists_x \forall_y F(x, y, w)$. Now $w \in L$ can be recognized by a red–green TM $\mathcal{M}$ that operates as follows. The machine starts in red, and carries out the following phases for all x in order, one after the other:

> Phase x:
>
> – switch to green
> – check whether $\forall_y F(x, y, w)$ by going through all possible strings y in order and evaluating $F(x, y, w)$.
> – if a y is encountered for which $F(x, y, w)$ does *not* hold, then stop checking, switch to red, exit this phase (and thus move on to the next phase, with the next value of x).

Obviously, there exists an x such that for all y predicate $F(x, y, w)$ is satisfied, if and only if the machine will only make a finite number of mind changes before settling in one phase forever (thus in green). It follows that, by definition, L is recognized by the red–green TM $\mathcal{M}$.

If $L \in \Delta_2$, then there exist recursive predicate $F(x, y, w)$ and $G(x, y, w)$ such that $w \in L \Leftrightarrow \exists_x \forall_y F(x, y, w)$ and $w \in \bar{L} \Leftrightarrow \exists_x \forall_y G(x, y, w)$. Now design a red–green TM $\mathcal{M}$ that operates as follows on input x. Again the machine starts in red, and carries out phases for all x in order, one after the other:

> Phase x:
>
> – switch to green
> – check whether $\forall_y F(x, y, w)$ by going through all possible strings y in order and evaluating $F(x, y, w)$.
> – if a y is encountered for which $F(x, y, w)$ does *not* hold, then stop checking F and switch to red,
> – check whether $\forall_y G(x, y, w)$ by going through all possible strings y in order and evaluating $G(x, y, w)$
> – if a y is encountered for which $G(x, y, w)$ does *not* hold, then stop checking (and thus move on to the next phase and switching to green with the next value of x).

Because $w \in L$ or $w \in \bar{L}$, there must be some finite x such that $\forall_y F(x, y, w)$ or $\forall_y G(x, y, w)$. Consequently, $\mathcal{M}$ must enter a phase eventually in which it stays put, necessarily after at most finitely many mind changes, recognizing w when $w \in L$ and stabilizing in red when $w \in \bar{L}$. Thus L can be accepted by a red–green machine. □

Theorem 1 shows the extent of red–green machines, but is also helpful in proving concrete sets of strings to be in Σ_2 (Δ_2).

Example 1. For strings w, let M_w denote the TM with code w. Let $T_0(n), T_1(n), \ldots$ be a computable enumeration of time bounds. Then $\{w | M_w$ works in time $T_k(n)$ for some $k\} \in \Sigma_2$. (This generalizes the example due to Hajék [10] for the cases of polynomial time with/without fixed degree.) Simply design a red–green TM $\mathcal{M}$ which on input w operates as follows. $\mathcal{M}$ starts in red and carries out phases $k = 0, 1, \ldots$ in which it (a) switches to green, (b) generates the code for computing $T_k(n)$ by enumeration, and (c) simulates M_w on all inputs x one after the other for $T_k(|x|)$ steps until it encounters an x for

which M_w needs more steps. If the latter case does not arise, $w \in L$ and $\mathcal{M}$ indeed accepts it. If the latter case arises, machine $\mathcal{M}$ stops checking, switches to red, and moves to phase $k+1$. This proves that L is red–green recognizable, thus in Σ_2.

2.3. Nondeterminism

We now turn to nondeterministic red–green computing. Can it be characterized as well and, does it give advantages over deterministic computing?

A very effective way to represent the computations ('runs') of a nondeterministic red–green TM $\mathcal{M}$ on input w is given by the *computation tree* of $\mathcal{M}$ on w. This is the infinite tree T (we omit $\mathcal{M}$ and w as explicit subscripts) in which (a) every node corresponds to the configuration after a finite run of $\mathcal{M}$ on w, (b) the root corresponds to the initial configuration of $\mathcal{M}$ on w, and (c) if $\mathcal{C}$ is the configuration associated to any node v, then v's sons uniquely and precisely correspond to a successor configuration of $\mathcal{C}$ after applying an option of the nondeterministic next move to $\mathcal{C}$.

Because we assumed nondeterministic machines always to have one or two options available (only) in any nondeterministic move, T is an infinite tree with all nodes having one or two sons only. Also, because $\mathcal{M}$ is a red–green TM, we can associate a unique colour to every node of $T = T_{\mathcal{M},w}$: red if the node corresponds to a configuration with a red state, green if it corresponds to a configuration with a green state.

Fact. *T has no leaves at finite depth. The infinite paths in T from the root down correspond precisely to the possible runs of $\mathcal{M}$ on w. The string w is recognized if and only if T has an infinite path from the root down that is all green from some finite level onward.*

The basic observation for nondeterministic red–green TMs is the following. In the proof we use the notion of a *cut* of T.

Definition 8. A cut of T is any finite set of *red* nodes such that (a) no node in the set is an ancestor of another, and (b) every path from the root down must hit a node in the set.

A cut separates T in a part above and below the cut. Viewing every node in the cut as the root of the infinite subtree below it, the nodes of a cut 'span' the entire tree below the cut.

Theorem 3. *Let L be the set of strings recognized by a nondeterministic red–green TM. Then L can also be recognized by a deterministic red–green TM.*

Proof (*Sketch*)**.** Let L be recognized by nondeterministic red–green TM $\mathcal{M}$, and w some input. The idea is to perform a combined breadth-first (BFS) and depth-first search (DFS) of the computation tree of $\mathcal{M}$ on w. The DFS is used for "in-depth" inspection of subtrees with a green root and discovering possible infinite green paths in them, whereas the BFS is used for inspecting the remaining red nodes.

More precisely, design a deterministic red–green TM $\mathcal{N}$ as follows. $\mathcal{N}$ works in successive phases $j = 1, 2, \ldots$ for as long as it goes. At the beginning of phase j, $\mathcal{N}$ has marked a cut of T (e.g. as a linked list of nodes from the left end of T to the right end), with the part above the cut finite and explored and the part below the cut still unexplored. The initial condition is certainly satisfied at the beginning of the initial phase ($j = 1$), where we assume that $\mathcal{N}$ start with the cut consisting just of the root of T (a red node). If the machine has gotten to phase j, let the cut it got to consist of nodes $v_1, v_2, \ldots$ and the machine be in *red*. The machine now proceeds as follows:

Phase j:

- switch to *green*.
- inspect the nodes $v_1, v_2, \ldots$ from left to right and try to construct a next lower cut of the tree as follows. Say we have constructed the beginning part $w_1, w_2, \ldots, w_s$ of the next cut. Let v_t (some $t \geq 1$) be the next node of the current cut to be considered.
- expand v_t, say into nodes $v_{t,0}$ and $v_{t,1}$. If $v_{t,0}$ is red, append it to the list of the next cut and consider $v_{t,1}$.
- if $v_{t,0}$ is green, it may be the starting node of an infinite green path. We call the DFS part to try and find it, while staying in *green*:
  - expand (the tree below) $v_{t,0}$ by DFS: whenever a green node is encountered, the DFS continues to expanding to lower nodes, and when a red node is encountered the DFS backs up and the red node is appended to the list of the next cut.
  - if the DFS backs up to $v_{t,0}$ again and $v_{t,0}$ has no unexplored sons anymore, then the DFS ends.
- handle $v_{t,1}$ in the same way.
- if (the subtree below) $v_{t,1}$ is handled, proceed with the BFS to v_{t+1} if it exists. If the cut is depleted, switch to *red* (!) and exit the phase.

If a green node e.g. $v_{t,0}$ in the description above is expanded in a phase, two things can happen: either there is an infinite green path starting at $v_{t,0}$ or there is not. In the former case, the search will guide $\mathcal{N}$ to the lexicographically first green path and the DFS will continue forever, staying in green despite that some finite branches ending at a red node may get entered every once in a while). In the case that there is no infinite green path starting at $v_{t,0}$, the DFS will eventually end and the BFS process will proceed to the next node of the cut. It follows that during phase j, either an infinite green path is found and $\mathcal{N}$

stays in this phase in green forever, or the phase is completed in finite time, with a switch to red at the end and a new deeper cut of T constructed as starting condition for phase $j+1$. The entire phase is carried out in green and only if the phase ends is finite time, a mind change is taking place in entering the next phase.

If $w \in L$, then in some phase j an infinite green path will be found and $\mathcal{N}$ indeed recognizes w after having made at most finitely many mind changes. If $w \notin L$ the latter cannot happen. In this case, every phase will end after finitely many steps (and at least one mind change) and $\mathcal{N}$ will go through next phases without end, with infinitely many mind changes.

We conclude that $\mathcal{N}$ is a deterministic red–green TM that recognizes L. □

3. Mind change complexity

In this Section we study (a) the computational potential red–green TMs that make at most a constantly bounded number of mind changes, and (b) the connection between red–green TMs and relativistic computing. We will also argue that e.g. $P^{mind} \subsetneq \Delta_2$.

3.1. Bounding the number of mind changes

We will show that red–green TMs that recognize (thus accept) their inputs after at most k mind changes, for some fixed constant k, precisely characterize the classes Σ_k^{-1} of the *Boolean Hierarchy* due to Ershov [5]. Putnam ([18], Theorem 2) gave a similar characterization by means of his *k-trial predicates*, in purely functional terms only (cf. [15], p. 374).

Note that $\Sigma_k^{-1} \subseteq \Delta_2$ for every k, and that $\cup_{k=0}^{\infty} \Sigma_k^{-1}$ is the Boolean closure of the class of recursively enumerable sets. The sets in a class Σ_k^{-1} are also called the k-r.e. sets and can be described as follows.

Definition 9. An k-r.e. set is any set A that can be written as the symmetric difference of k r.e. sets. Alternatively, A is k-r.e. if there are k r.e. sets $A_1, \ldots, A_k$ such that for any string w: $w \in A$ if and only if $w \in A_i$ for an *odd* number of indices i.

Theorem 4. *For any $k \geq 1$, L is the set of strings accepted by a red–green TM within k mind changes if and only if L is an k-r.e. set.*

Proof. ($\Rightarrow$) Let L be accepted by a red–green TM $\mathcal{M}$ within at most k mind changes. We construct k r.e. sets $A_1, \ldots, A_n$ such that $w \in L$ if and only if w belongs to an odd number of A_is.

Design a TM $\mathcal{N}$ that starts with each $A_i = \emptyset$ and generates the strings $w \in \Sigma^*$ one after the other, adding each w to some of A_i's as needed. The basic idea is this. For each input w, $\mathcal{N}$ runs a simulation of machine $\mathcal{M}$ on w. When $\mathcal{M}$ switches over to a green state, we add w to A_1. Each time when $\mathcal{M}$ makes another mind change, we add w to the next A_i. This continues until we add w to A_k the latest. The construction guarantees that all A_i are recursively enumerable.

Note that when the computation of $\mathcal{M}$ on w will converge to green (red) within the allowed number of mind changes, it means that $w \in L$ ($w \notin L$) and w has been assigned to an odd (even, respectively) number of sets. This shows that L is the symmetric difference of the A_i, i.e. that L is a n-r.e. set.

($\Leftarrow$) Now let L be k-r.e. set, and let $A_1, \ldots, A_k$ be r.e. sets such that $w \in L$ if and only if w belongs to an odd number of A_is. Let $\mathcal{N}_1, \ldots, \mathcal{N}_{k+1}$ be the $k+1$ TMs recognizing the respective r.e. sets.

Construct a red–green TM $\mathcal{M}$ that works as follows. On input w, $\mathcal{M}$ starts a simultaneous simulation of $\mathcal{N}_1, \ldots, \mathcal{N}_{k+1}$ on this input. Depending on the termination of any of the $\mathcal{N}_i$, $\mathcal{M}$ does the following. As usual it starts computing in red. Whenever a $\mathcal{N}_i$ terminates (signalling that $w \in A_i$), $\mathcal{M}$ switch colour and continues simulating (or just converging if all $\mathcal{N}_i$ happened to have terminated). Note that $\mathcal{M}$ can make at most k mind changes and, as strings $w \in L$ belong to an odd number of A_i and $w \notin L$ do not, $\mathcal{M}$ ends up in green precisely when $w \in L$. □

Thus, the computational potential of red–green TMs that make at most a constantly bounded number of mind changes is completely characterized, namely by the level sets of the Boolean Hierarchy. Theorem 4 also follows from Lemma 1 by characterizing the Σ_k^{-1} sets as sets of the form $(A_1 - A_2) \cup (A_2 - A_3) \cup \cdots \cup (A_k - A_{k+1})$, for recursively enumerable sets $A_1, \ldots, A_{k+1}$. The following observation is immediate.

Theorem 5. *For any odd $k \geq 1$ there exist sets in Δ_2 which can be accepted by a red–green TM within $k+1$ mind changes but not by any red–green TM within k mind changes.*

3.2. About P^{mind} and computing with oracle calls

Having characterized the power of red–green TMs that only make a up to a constant number of mind changes, we turn to the more general case. First consider the case of a polynomially bounded number of mind changes.

Theorem 6. *For every computable function f, $MIND[f] \subsetneq \Delta_2$. In particular, $P^{mind} \subsetneq \Delta_2$.*

Proof. Red–green TMs that recognize their strings in a computably bounded number of mind changes, can keep a count of their number of mind changes and observe whether this number goes over the bound or not. Thus, they can be assumed to work as acceptors. Now apply Theorem 2 (ii). Strict inclusion follows by a diagonal argument similar to that in Theorem 5. □

Relativistic computing

Several years ago a type of computing was proposed that allowed the use of explicit 'infinite time subroutines' and result signals of these calls returning in 'finite time'. The model was motivated by a possible physical reality according to general relativity theory and was called *relativistic computing* [6]. We showed that relativistic computers accept exactly the Δ_2 sets and developed a complexity theory for these machines when operating within polynomial space [29] (see also [28]). Theorem 2(ii) immediately gives us a concrete realistic 'model' equivalent to relativistic computing.

Theorem 7. *Relativistic computers are equivalent to red–green TM acceptors: both accept precisely the Δ_2 sets.*

Relativistic computers are almost by definition equivalent to (oracle) TMs that are allowed to carry out at most finitely many *yes/no-calls* to the Halting Problem. Thus Theorem 7 is essentially equivalent to Post's theorem ([15], Proposition IV.1.16) and expresses a characterization similar to Shoenfield's Limit Lemma ([15], Proposition IV.1.17), now in the far more flexible and expressive formalism of red–green acceptors. It was shown by Beigel [1] that TMs with $n+1$ calls to the Halting Problem compute a strictly larger set of partial functions than TMs with n calls to the Halting Problem.

Relating mind change and oracle call complexity

Theorem 7 resolves the standing question to characterize relativistic computing by a machine model that fits more realistically in the existing paradigm of infinite computation. In fact, one can show that the characterization can be proved directly in machine terms, with the advantage that it leads to a direct proof of Post's theorem that explicitly links the complexity of the models. The proof relates (connects) the realistic measure of converge time complexity in red–green TMs to the not-so-realistic measure of oracle call complexity in TMs that are recursive in **K** (thus relativistic computers).

Theorem 8. *The oracle call complexity of relativistic computers and the mind change complexity of red–green TM acceptors are linearly bounded in each other.*

The interesting conclusion we can draw is that relativistic computing essentially coincides with red–green acceptors in capabilities. Thus one can study the former by studying the latter which, in a sense, allows for a more direct computational analysis.

It follows that red–green TMs are the 'infinitistic' realization of relativistic computing, placing the latter clearly in the spectrum of hypercomputation.

4. Nondeterminism and mind change complexity

We now return to the study of nondeterministic red–green TMs. Recall that the machines recognize exactly the Σ_2 sets. What can one say more in particular about nondeterministic red–green TMs and their behaviour? Again some familiar recursion-theoretic facts will reappear in the new computational context of these machines and their analysis.

4.1. Simulating nondeterminism

In Section 2.3 we proved that nondeterministic red–green TMs are no more powerful than their deterministic version. We left it open to relate the mind change complexity of the deterministic simulation to that of the nondeterministic machine. Here we add a further observation.

A nondeterministic red–green TM, operating on input w and accepting this input by admitting a computation on w which converges to all green in finite time, can in theory admit every other type of computation on w as well: it may admit computations that do not stabilize at all (thus making an unbounded number of mind changes as the computation proceeds), and it may in fact even admit computations that eventually stabilize in red (which thus does not necessarily say anything about acceptance of the input). We now consider nondeterministic red–green TMs which do *not* admit the latter type of computations when accepting a string.

Definition 10. A nondeterministic red–green TM $\mathcal{M}$ is called *non-ambiguous* if on all recognized inputs w, $\mathcal{M}$ does not admit any computations that converge to all red. $\mathcal{M}$ is called *ambiguous* otherwise.

Non-ambiguous (nondeterministic) red–green TMs can never have accepting runs (converging to green) and 'rejecting' runs (converging to red) simultaneously, but accepting runs and divergent runs (with infinitely many mind changes) are allowed. We do not care how the machine behaves on inputs that are not recognized.

Theorem 9. *Let L be the set of strings recognized by a non-ambiguous nondeterministic red–green TM within mind change complexity $MC(w)$. Then L can be recognized by a deterministic red–green TM within the same mind change complexity $MC(w)$.*

Proof (*Sketch*)**.** Let L be recognized by a non-ambiguous nondeterministic red–green TM $\mathcal{M}$ and w some input. As in the proof of Theorem 2 we will construct a deterministic red–green TM $\mathcal{N}$ that explores the computation tree $T = T_{\mathcal{M},w}$ from the root down in a particular way.

Let us first assume that w is a recognized input, i.e. T contains an infinite path that is all green from some finite level on down but, by non-ambiguity, T does not contain any path that is all red from some level onward. The latter implies that along every path, every consecutive stretch of red nodes is necessarily finite. This suggest the following modification of the procedure described in Theorem 2.

Again $\mathcal{N}$ will reconstruct and explore T in phases, in every phase 'moving' a cut of T to a next cut of T, with the possibility that in doing so the DFS hits on an infinite green path. However, this time $\mathcal{N}$ will have red and green phases alternating and phases have a slightly different initial condition. $\mathcal{N}$ begins in a red phase, with a (red) cut consisting of the root of T (only). Then the phases proceed as follows:

- red phases begin with a red cross cut C. The nodes of the cut are expanded in red in BFS/DFS fashion, where nodes are expanded all the way until green nodes are reached along the entire search frontier. Green nodes will eventually be reached along all search paths in finitely many steps, by the non-ambiguity of $\mathcal{M}$. When all (red) nodes are explored from left to right, the red phase ends. Accumulating the green nodes along the search front in a list from left to right will result in another cut C', this time consisting of green nodes only. When the red phase has ended, $\mathcal{N}$ will switch to the next phase, which will be a green phase. (Observe that the tree from C 'down to but not including' C' consists entirely of red nodes and that it is in a sense the maximal such region from C on downward.) Entering a green phase is accompanied by switching to green.
- green phases begin with a green cross cut C. The nodes of the cut are expanded in green in BFS/DFS fashion, in exactly the same way as described in the proof of Theorem 2. Now the nodes are expanded up to the red nodes reached and this leads to another cut C', unless the process hits on an infinite green path which keeps the DFS going indefinitely (in green). If we do not hit on an infinite green path in this phase, the phase will end in finite time, with a new cut C' now consisting entirely of red nodes. $\mathcal{N}$ will then switch to red and to the next phase, which will be a red phase again. (Observe that if the phase is finite, the tree from C 'down to but not including' C' consists entirely of green nodes and that it is in a sense the maximal such region from C on downward. If the phase is not finite, it will essentially trace the lexicographically first infinite green path from some node in C downward.)

Clearly $\mathcal{N}$ will recognize w precisely in a green phase, sufficiently far into the simulation. Considering any recognizing path π for w down T and the succession of red and green segments on it, one observes that the phases of the simulation reconstruct and include the successive segments as they alternate until (or unless) a phase becomes infinite because it hits on an infinite green path. It follows that the number of mind changes occurring during the simulation is precisely $MC(w)$. The entire procedure can be coded for a red–green TM.

Now assume that w is not a recognized input. Consider the procedure as described above. Clearly there is no chance now to hit on an infinite green path, i.e. green phases will always be finite now. However, the simulation does not necessarily give an infinite alternation of red and green now: it can now happen that the simulation hits on an infinite red path while in a red phase, which will make the phase infinite and the computation converge in red. But in either case it implies that w is not recognized. Thus the recognition process is correctly simulated. □

Several conditions other than non-ambiguity can be given which allow for a simulation result similar to Theorem 9.

4.2. Complexity consequences

What is the real impact of the machine-based notion of nondeterminism for the case of red–green TMs? We first prove a simple, but crucial observation that shows why the unrestricted notion is crushing the complexity concept we have been using.

Lemma 3. *Every set of strings $L \in \Sigma_2$ can be recognized by a nondeterministic red–green TM within one mind change.*

Proof. Let $L \in \Sigma_2$. This implies that there is a recursive predicate $F(x, y, w)$ such that $w \in L \Leftrightarrow \exists_x \forall_y F(x, y, w)$. To recognize strings w of L one can thus design a nondeterministic red–green TM $\mathcal{M}$ that operates as follows. On input w, the machine first 'nondeterministically' generates some arbitrary string x in red, it then switches to green and starts verifying by enumeration that for every x indeed $F(x, y, w)$ holds. If the predicate always holds, then $\mathcal{M}$ will converge in green and recognizes w correctly, after only one mind change. If the machine encounters an x for which $F(x, y, w)$ does *not* hold, then the machine switches back to red and stays in red forever. $\mathcal{M}$ clearly nondeterministically recognizes L. □

Corollary 1. *For every function g with $g(w) \geq 1$, $ND\_MIND[g] = \Sigma_2$.*

Comparing Lemmas 2 and 3, it follows that nondeterminism precisely causes the step up from Σ_1 to Σ_2. Mind change complexity is a way to fill the computational gap only in the deterministic case. We draw some straightforward conclusions from the elaborations.

Theorem 10. *For every function g, $MIND[g] \subsetneq ND\_MIND[1]$. In particular $P^{mind} \subsetneq NP^{mind}$.*

Proof. By Theorem 2, $MIND[g] \subset \Delta_2$. By Lemma 3, $ND\_MIND[1] = \Sigma_2$. The result follows. □

Corollary 2. *There is no computable function f such that for all sets of strings L, if L is recognized by a nondeterministic red–green TM within k mind changes, then L can be recognized by a deterministic red–green TM within $f(k)$ mind changes.*

Another observation can be made, based on the analysis in Section 4.1. It gives a concrete fact for the complexity of nondeterministic machines that recognize sets of strings beyond P^{mind} in a polynomially bounded number of mind changes. A similar fact can be stated for nondeterministic red–green TMs recognizing set of strings in $\Sigma_2 - MIND[f]$ within $f(|w|)$ mind changes

Corollary 3. *Any nondeterministic red–green TM recognizing a set of strings in $NP^{mind} - P^{mind}$ within polynomially many mind changes is necessarily ambiguous, i.e. must have inputs w on which it has both accepting runs (converging to green) and rejecting runs (converging to red).*

Proof. Consider any language $L \notin P^{mind}$. Suppose L could be recognized by a non-ambiguous, nondeterministic red–green machine. By Theorem 9, L can also be recognized deterministically within polynomially many mind changes. By Theorem 6, $L \in P^{mind}$. Contradiction. □

5. Conclusions

What would the concept of computability look like if it were invented today? In all likelihood a step-level computational mechanism would be modelled in some way equivalent to classical TMs, if we may believe the Church-Turing thesis. But computations as conceived of and programmed today, have a number of features that extend beyond pure calculation. In this study we have specifically addressed the issue of perpetuating computations. How can they be modelled and to what theoretical framework do they lead? Models of unbounded computation have been studied in the past, often obtained by allowing finite machines to compute 'forever' and defining the behaviour over infinite input strings (ω-sequences) in some way. We have tried to start from first principles, and on finite inputs.

The model of unbounded computation we presented, derives from an idealized notion of multi-process computation. The model leads to red–green Turing machines and several measures to assess their complexity. Red–green TMs allow an escape from finite to unbounded computation in a way that is both credible and amenable to computational analysis. At the same time there may be didactic advantages to the model, compared to other approaches.

The notion of mind change complexity gives a way to measure unbounded computations and enables a characterization of the 'jump' in computational power achieved by going to unboundedness, a jump from the first to the second level of the *Arithmetical Hierarchy*. The model can be seen as a concrete motivation to consider problems up to Δ_2 or even Σ_2 computable 'by an unbounded process'. The extension to higher levels is tempting e.g. by superimposing *alternation*. This is open for further study although it would lead away by yet another step from the basic principle of computation as an unbounded process studied here. Connections between computational models and the higher levels of the Arithmetical Hierarchy have been found before (see e.g. [13,22]) but the model of red–green TMs seems to be more insightful.

Interestingly, the model of red–green TMs subsumes several existing recursion-theoretic models in the literature such as *trial and error predicates* [18] and *limiting recursion* [7,8]. The model can even be related to Turing's fundamental 1936 paper, by interpreting it as the non-terminating version of Turing's circular a-machines. The model gives a concrete computational context and allows a complexity-theoretic analysis. For example, the model allows for a re-appraisal of the Kleene-Post characterization of Δ_2 and Σ_2 and a programmer's proof of it using red–green machines which also gives a quantitative relationship between the oracle call complexity of the former and the mind change complexity of the latter. We have specifically exploited the model to define the notion of nondeterminism in unbounded computation, showing that it can always be simulated deterministically but with an anomalous effect on mind change complexity. The $P^{mind} \neq NP^{mind}$ result is an example for the different intuitions in (nondeterministic) red–green computation. Nondeterministic red–green TMs that recognize sets in polynomially many mind changes that nonetheless do *not* belong to P^{mind}, are necessarily 'ambiguous' in the sense defined in Section 4.

An additional benefit of red–green TMs is that they provide a concrete implementation of the notion of 'relativistic computation', proposed a number of years ago [6]. It was shown before that relativistic computation essentially is computation at the level of Δ_2 [29,28] but it remained open to determine a realistic model for it in the realm of hypercomputation. Red–green computation is a possible answer. In an extension of our current study we have defined a higher-order version to capture the entire Arithmetical Hierarchy in feasible term of hypercomputability.

The further analysis of red–green TMs might involve not only a further study of mind change complexity and other measures, but also of a time-evolving notion of space complexity. The most useful features of red–green TMs seem be those that enable a quantization of the unbounded behaviour in finitistic terms.

Acknowledgements

The first author's research was partially supported by the Netherlands Institute for Advanced Study in the Humanities and Social Sciences (NIAS) and the Lorentz Center for the Sciences, Leiden, The Netherlands. The second author's research was partially supported by the institutional research plan AV0Z10300504 and the Czech National Science Foundation grant No. P202/10/1333.

References

[1] R. Beigel, Query-limited reducibilities, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, 1995.
[2] A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, Alternation, J. ACM 28 (1) (1981) 114–133. 28:1 (1981) 114–133.
[3] R.S. Cohen, A.Y. Gold, ω-computations on Turing machines, Theoret. Comput. Sci. 6 (1) (1978) 1–23.
[4] B.J. Copeland, Accelerating Turing machines, Minds Mach. 12 (2002) 281–301.
[5] Y.L. Ershov, A hierarchy of sets I, Algebra Logika 7 (1) (1968) 4774 (in Russian). Algebra Logic 7 (1) (1968) 2543 (English translation).
[6] G. Etesi, I. Németi, Non-Turing computation via Malament-Hogarth space–times, Internat. J. Theoret. Phys. 41 (2) (2002) 341–370.
[7] E.M. Gold, Limiting recursion, J. Symbolic Logic 30 (1) (1965) 28–48.
[8] E.M. Gold, Language identification in the limit, Inf. Control 10 (5) (1967) 447–474.
[9] O. Goldreich, Computational complexity — A conceptual perspective, Cambridge University Press, Cambridge, 2008.
[10] P. Hájek, Arithmetical Hierarchy and complexity of computation, Theoret. Comput. Sci. 8 (1979) 227–237.
[11] J.D. Hamkins, A. Lewis, Infinite time Turing machines, J. Symbolic Logic 65 (2) (2000) 567–604.
[12] J. Hintikka, A. Mutanen, An alternative concept of computability, in: J. Hintikka (Ed.), Language, Truth, and Logic in Mathematics, in: J. Hintikka Selected Papers, vol. 3, Kluwer Academic, Dordrecht, 1998, pp. 174–188.
[13] M. Hogarth, Deciding arithmetic using SAD computers, British J. Philos. Soc. 55 (2004) 681–691.
[14] S.C. Kleene, Recursive predicates and quantifiers, Trans. Amer. Math. Soc. 53 (1943) 41–73.
[15] P. Odifreddi, Classical recursion theory — The theory of functions and sets of natural numbers, in: Studies in Logic, vol. 125, Elsevier, Amsterdam, 1988.
[16] E.L. Post, Degrees of unsolvability: preliminary report, Bull. Amer. Math. Soc. 54 (1948) 641–642.
[17] P.H. Potgieter, Zeno machines and hypercomputation, Theoret. Comput. Sci. 358 (2006) 26–33.
[18] H. Putnam, Trial and error predicates and the solution to a problem of Mostowski, J. Symbolic Logic 30 (1) (1965) 49–57.
[19] H. Rogers Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, New York, 1967.
[20] B. Rovan, L. Steskal, Infinite computations and a hierarchy in Δ_3, in: S.B. Cooper, B. Löwe, A. Sorbi (Eds.), Computation and Logic in the Real World, Proc. Third Conference on Computability in Europe (CiE 2007), in: Lecture Notes in Computer Science, vol. 4497, Springer-Verlag, Berlin, 2007, pp. 660–669. see also: Infinite computations and a hierarchy in Δ_3 reconsidered, J. Logic Comput. 19 (1) (2009) 175–176.
[21] L.K. Schubert, Iterated limiting recursion and the program minimization problem, J. ACM 21 (3) (1974) 436–445.
[22] L. Staiger, ω-Languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol 3: Beyond Words, Springer-Verlag, Berlin, 1997, pp. 339–387. Chapter 6.
[23] W. Thomas, Automata on infinite objects, in: J. van Leeuwen (Ed.), Handbook of Theoretical computer Science, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 133–192.
[24] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, Proc. Lond. Math. Soc. 42 (1936–37) 230–265.
[25] A.M. Turing, Systems of logic based on ordinals, Proc. Lond. Math. Soc. 45 (1939) 161–228.
[26] J. van Leeuwen, J. Wiedermann, The Turing Machine paradigm in contemporary computing, in: B. Enquist, W. Schmidt (Eds.), Mathematics Unlimited — 2001 and Beyond, Springer-Verlag, Berlin, 2000, pp. 1139–1155.
[27] J. van Leeuwen, J. Wiedermann, The computational power of Turing's non-terminating circular a-machines, in: S.B. Cooper, J. van Leeuwen (Eds.), Alan Turing — His Work and Impact, Elsevier, 2012 (in press).
[28] P.D. Welch, The extent of computations in Malament-Hogarth spacetimes, British J. Philos. Sci. 59 (4) (2008) 659–674.
[29] J. Wiedermann, J. van Leeuwen, Relativistic computers and non-uniform complexity theory, in: C.S. Calude, et al. (Eds.), Unconventional Models of Computation, Proc. 3rd Int. Conference, UMC'02, in: Lecture Notes in Computer Science, vol. 2509, Springer-Verlag, Berlin, 2002, pp. 287–299.